Ethereum Virtual Machine Bytecode Analyzer

Guided Research

Alexander Roschlaub

September 26, 2019

Background

Security Analysis Framework for Smart Contracts

1. source code is not available (77% of unique smart contracts[1])
2. only visible at the level of bytecode (e.g. gas consumption)
3. optimizations performed by the compiler

[1] 01.2018 - EthIR

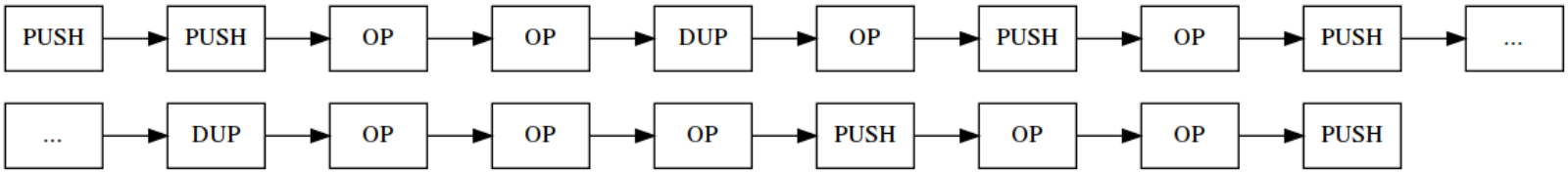
Outline

Bytecode Transformation(1): Vectorization

Input (Bytecode): 6080604052348015600f57600080fd5b50600436106045

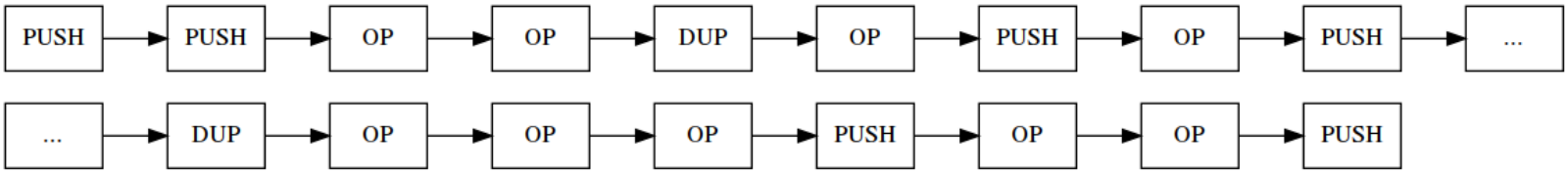
Bytecode Transformation(1): Vectorization

Input (Bytecode): 6080604052348015600f57600080fd5b50600436106045
Output: Objectsvector

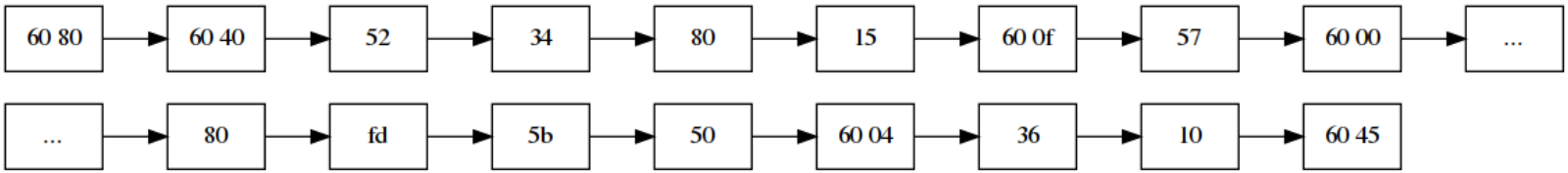


Bytecode Transformation(1): Vectorization

Input (Bytecode): 6080604052348015600f57600080fd5b50600436106045
Output: Objectsvector



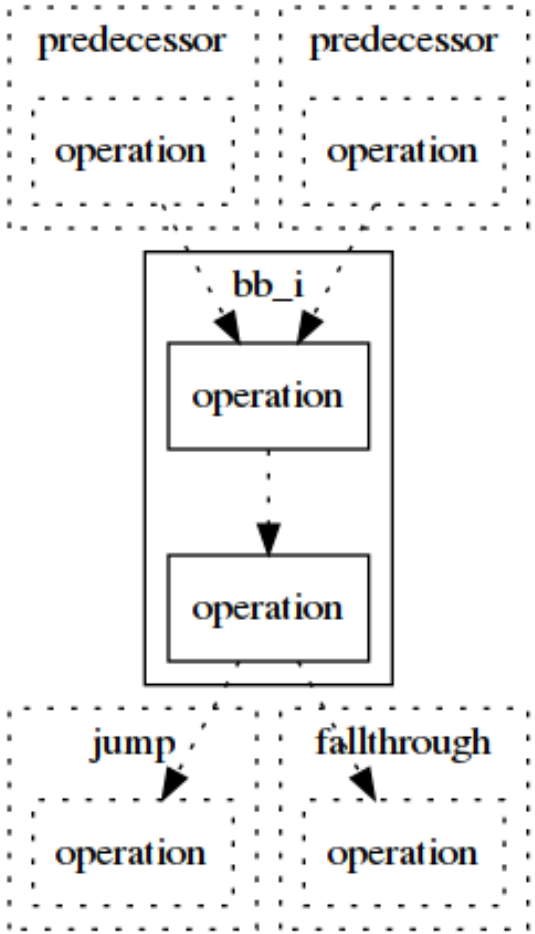
Filtering of Push Bytes:



Implemented: Operation (base case), Push, Swap, Dup

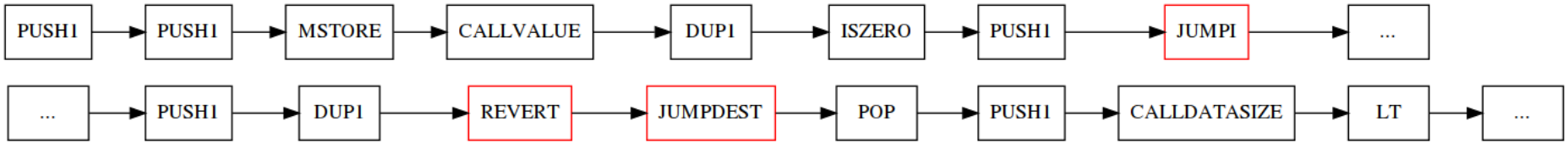
Definition: Basic Block

- $0 \ldots n$ predecessors
- content: $1 \ldots i$ operations
- $0 \ldots k$ successors (Jump, Fallthrough).



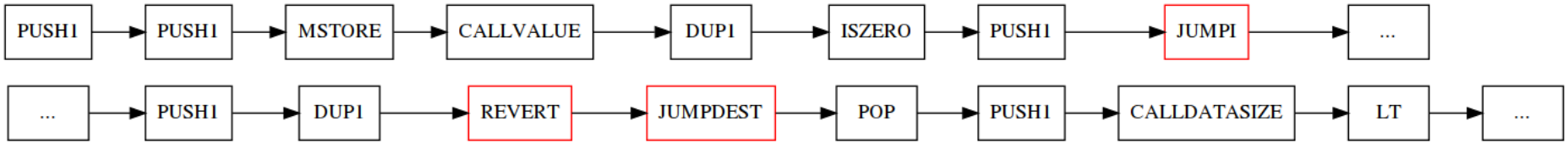
Bytecode Transformation(2): Basic Block Grouping

Input: Objectsvector

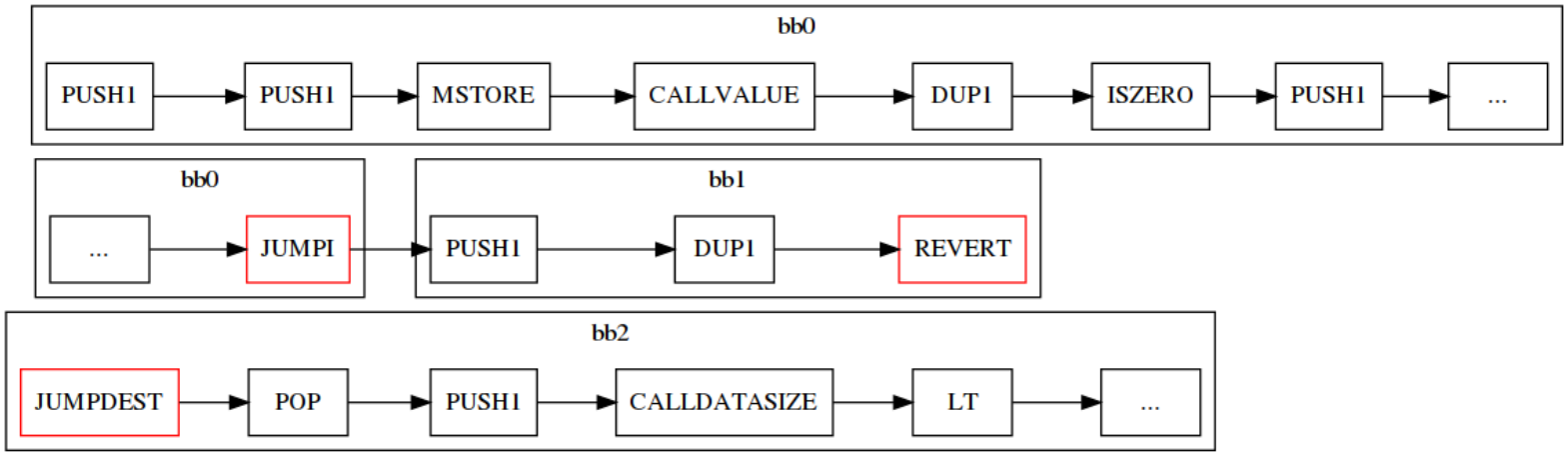


Bytecode Transformation(2): Basic Block Grouping

Input: Objectsvector

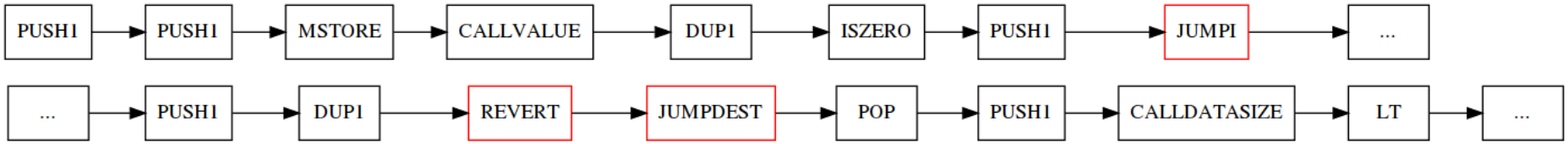


1. Group into Basic Blocks
2. Assign fallthrough successors

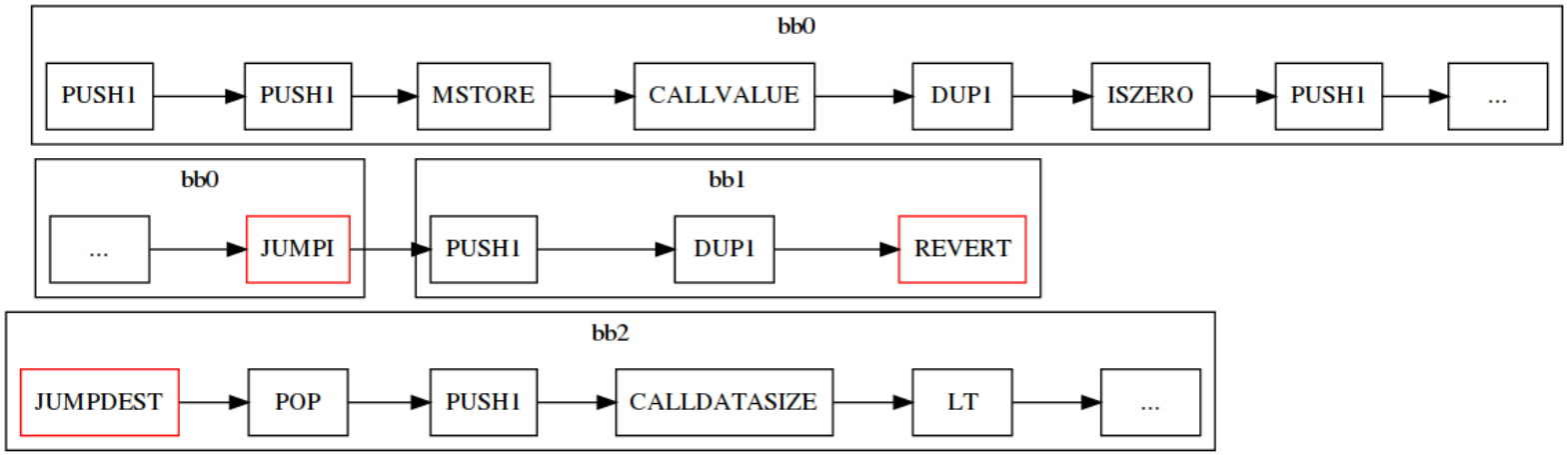


Bytecode Transformation(2): Basic Block Grouping

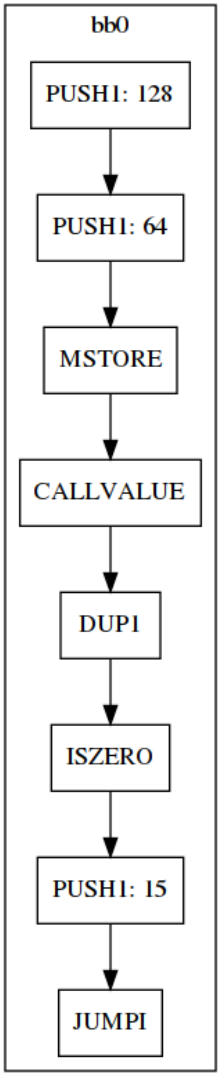
Input: Objectsvector



1. Group into Basic Blocks
2. Assign fallthrough successors
3. Problem: Position encoded jump destinations as stack items



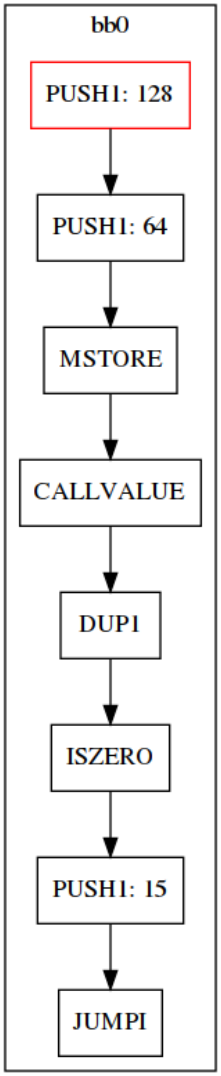
Basic Block Grouping: Adjust Jump Pointer



Each Operation has defined parameters:

- α: Number of elements pushed onto the stack
- δ: Number of elements popped from the stack
- 0 as dummy element (256 bit)

Basic Block Grouping: Adjust Jump Pointer



PUSH1:

- α: 1
- δ: 0

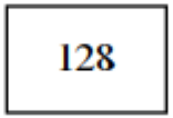
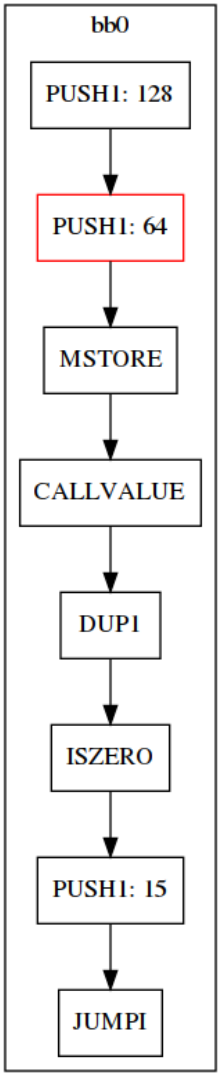


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



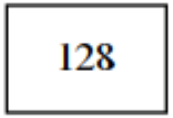


Figure: Stack State Before

PUSH1:

- α: 1
- δ: 0

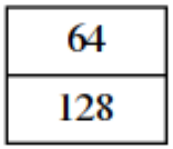
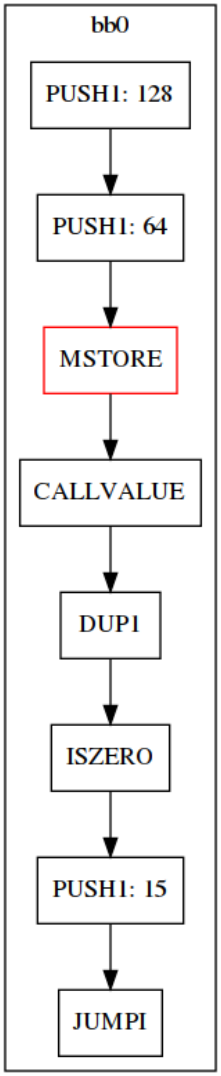


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



MSTORE:

- α: 0
- δ: 2

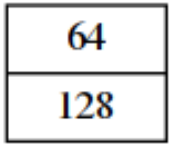


Figure: Stack State Before

Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer

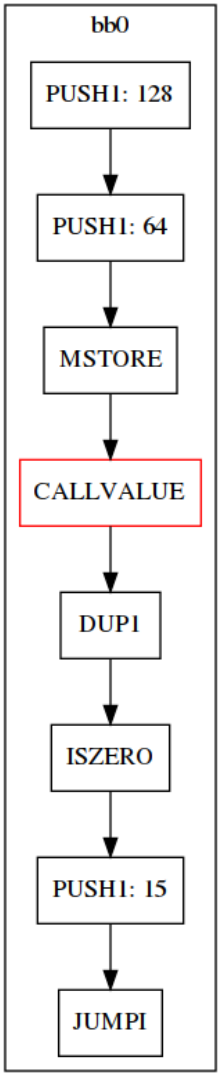


Figure: Stack State Before

CALLVALUE:

- α: 1
- δ: 0

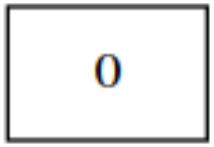
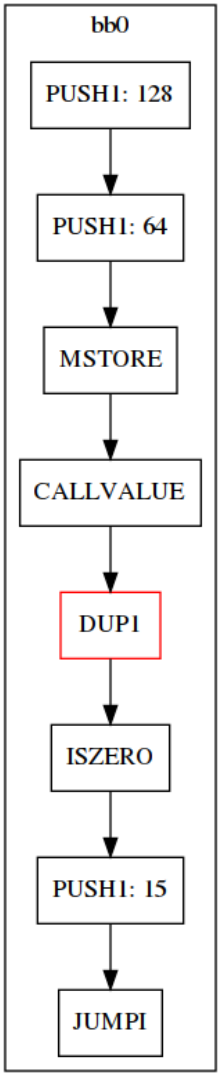


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



DUP1:

- α: 1
- δ: 1

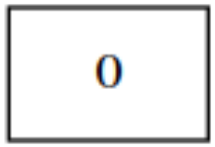


Figure: Stack State Before

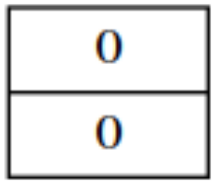
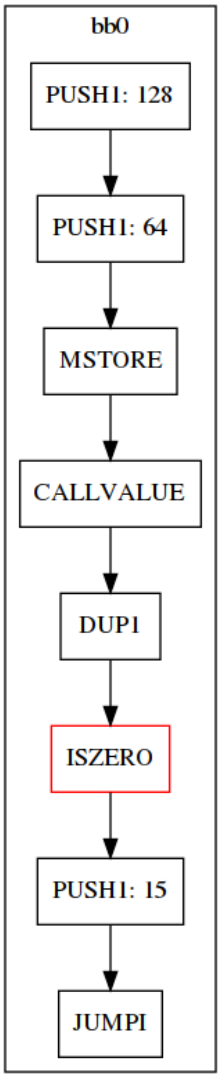


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



ISZERO:

- α: 1
- δ: 1

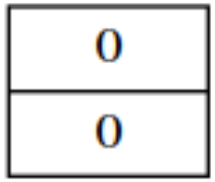


Figure: Stack State Before

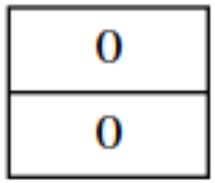
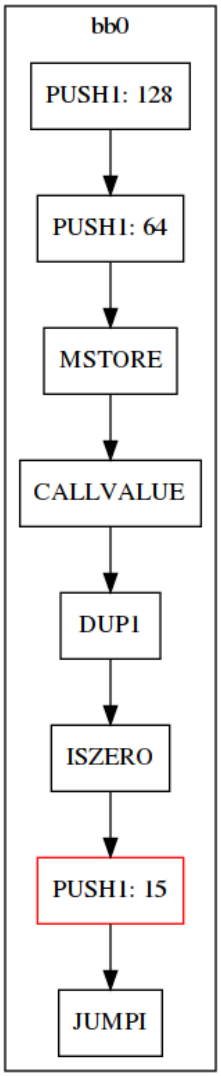


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



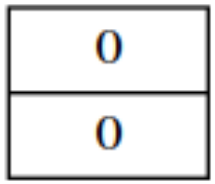


Figure: Stack State Before

PUSH1:

- α: 1
- δ: 0

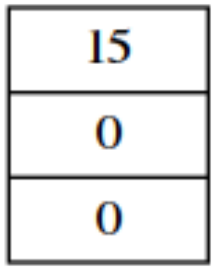
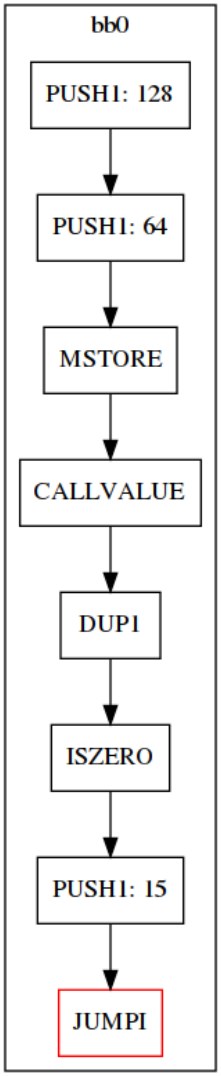


Figure: Stack State After

Basic Block Grouping: Adjust Jump Pointer



JUMPI:

- α: 0
- δ: 2

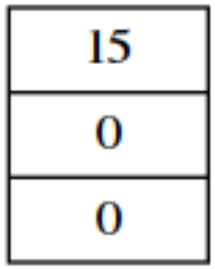


Figure: Stack State Before

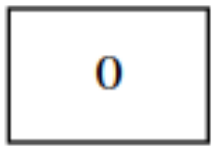
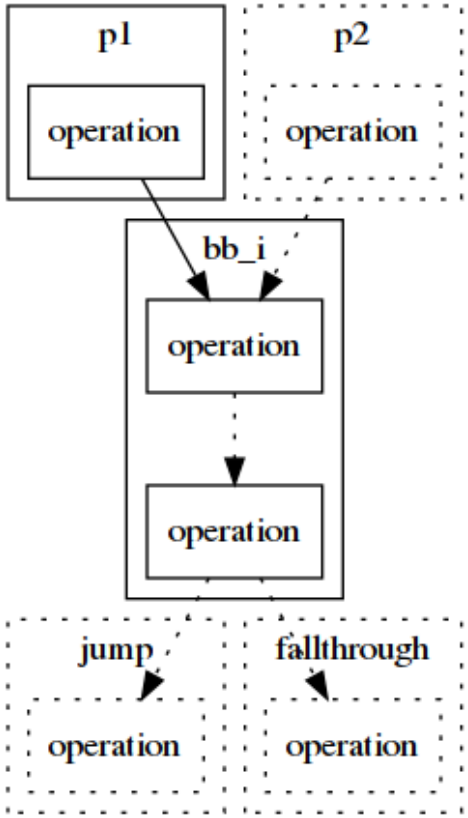


Figure: Stack State After

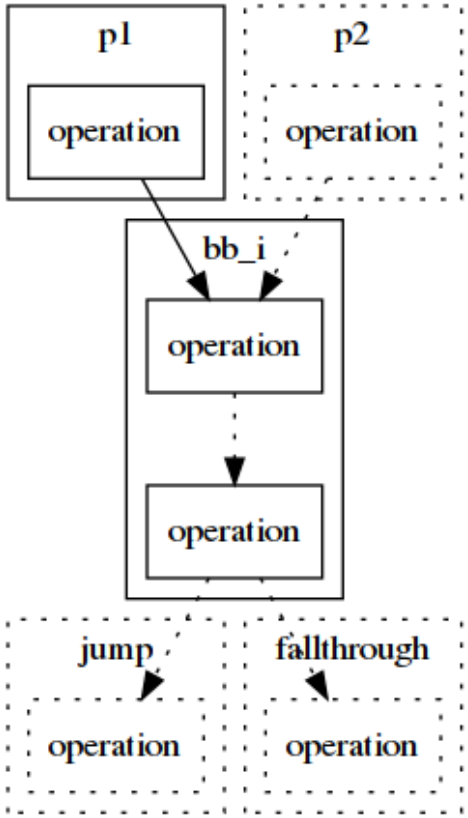
Basic Block Grouping: Merging Paths Problem

- Problem: Only the fallthrough predecessor is known beforehand



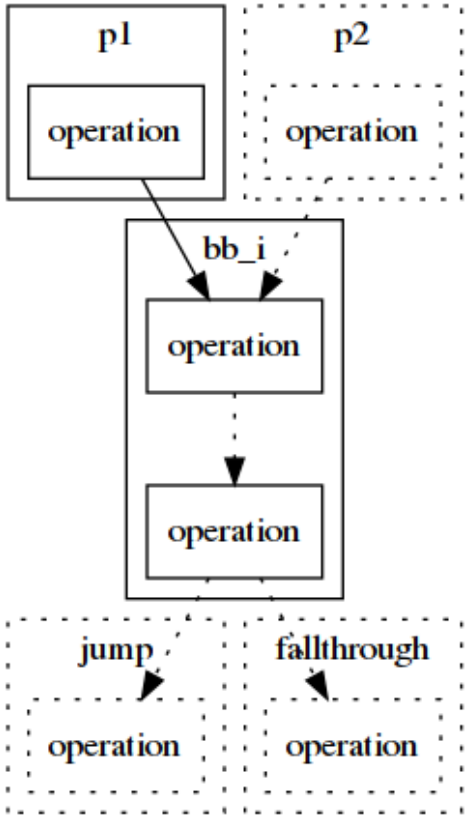
Basic Block Grouping: Merging Paths Problem

- Problem: Only the fallthrough predecessor is known beforehand
- Condition: Evaluation of jump positions only



Basic Block Grouping: Merging Paths Problem

- Problem: Only the fallthrough predecessor is known beforehand
- Condition: Evaluation of jump positions only
- Solution: Error output for duplicate assignments (Better: Fixpoint Iteration)



CFG vs optimized CFG

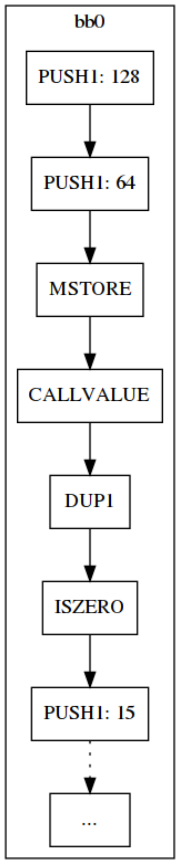


Figure: CFG(I)

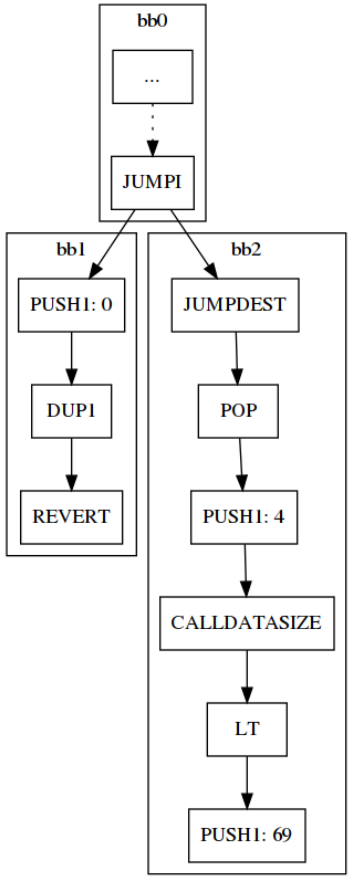


Figure: CFG(II)

CFG vs optimized CFG

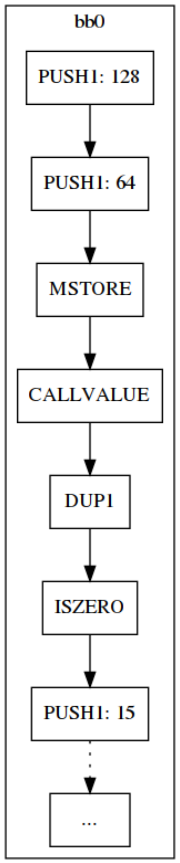


Figure: CFG(I)

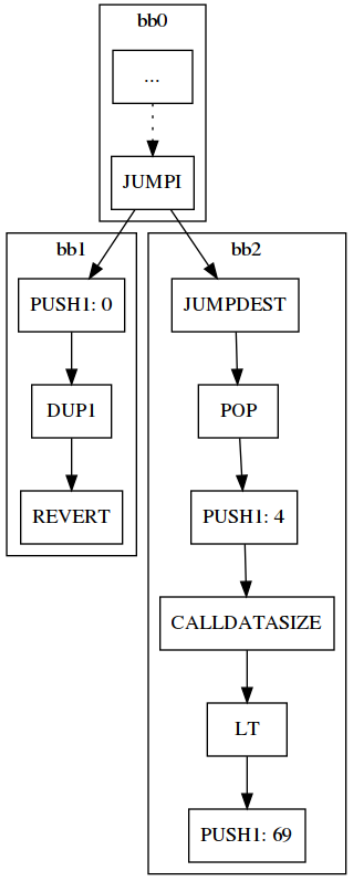


Figure: CFG(II)

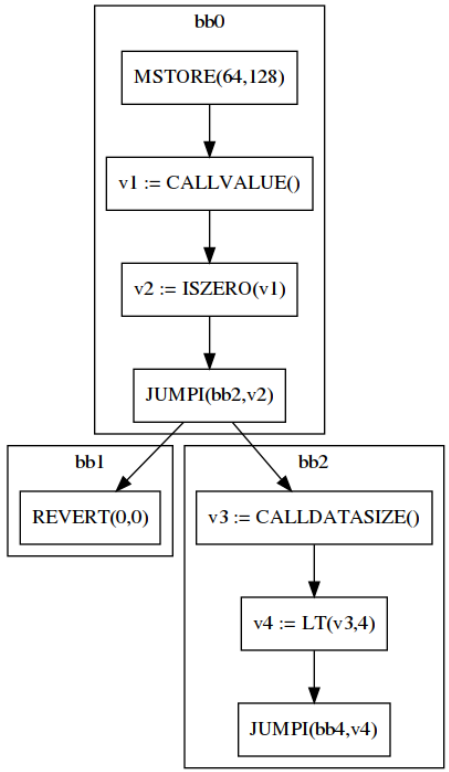
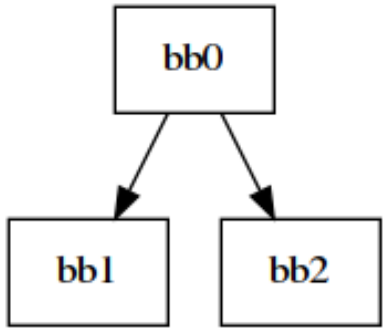


Figure: Optimized CFG

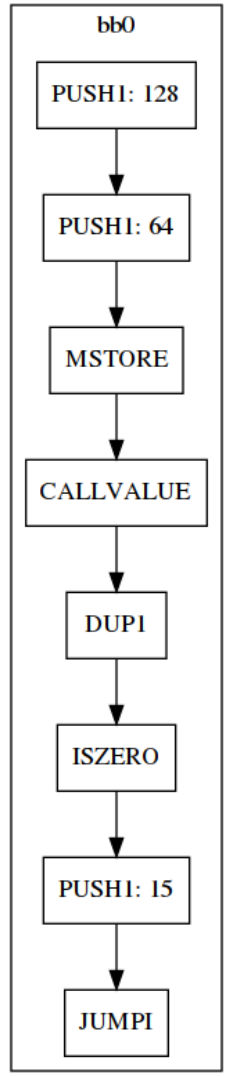
Stack Abstraction: First Idea

1. Replicate the Basic Blocks from the CFG with empty content



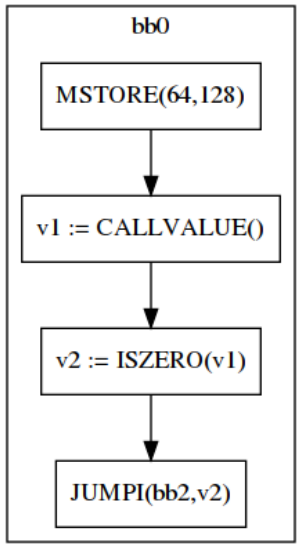
Stack Abstraction: First Idea

1. Replicate the Basic Blocks from the CFG with empty content
2. Recursive instantiation of Basic Blocks



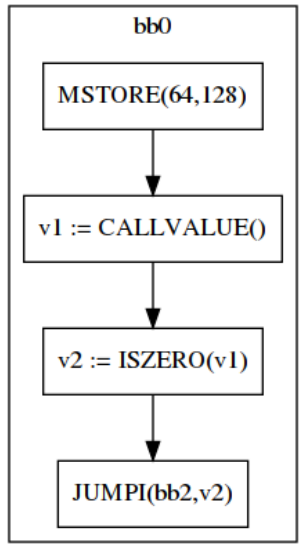
Stack Abstraction: First Idea

1. Replicate the Basic Blocks from the CFG with empty content
2. Recursive instantiation of Basic Blocks
   1. Transform the Operations into Instructions



Stack Abstraction: First Idea

1. Replicate the Basic Blocks from the CFG with empty content
2. Recursive instantiation of Basic Blocks
   1. Transform the Operations into Instructions
   2. Pass the stack to successors



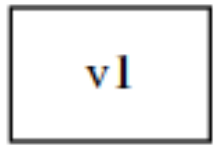


Figure: Stack state bb0

Merging Paths Problem

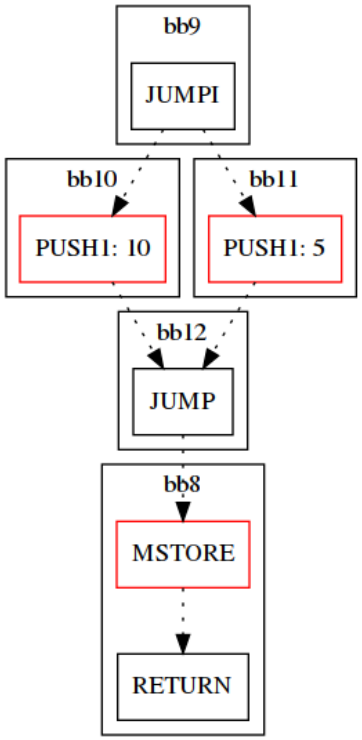


Figure: CFG

Merging Paths Problem

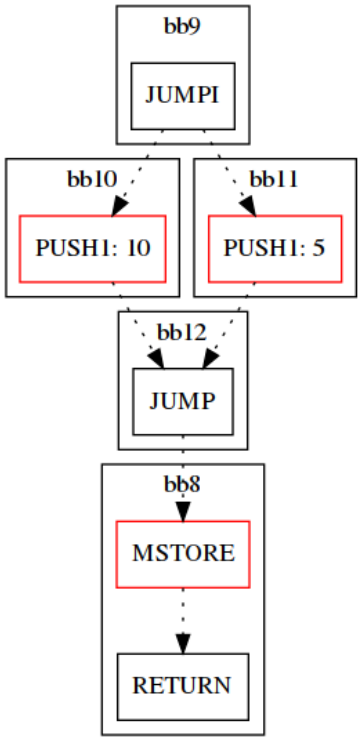


Figure: CFG

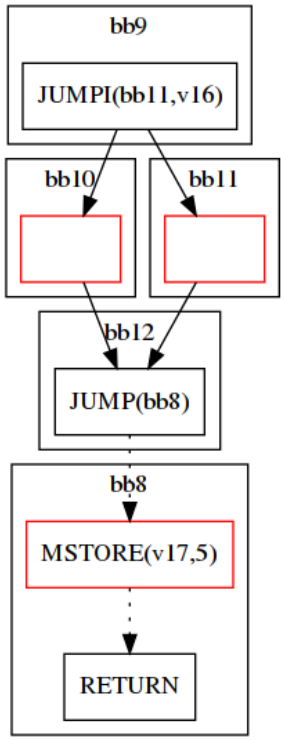


Figure: First Idea

Merging Paths Problem

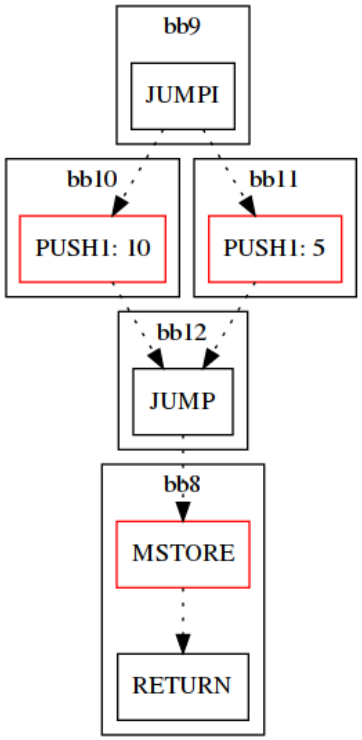


Figure: CFG

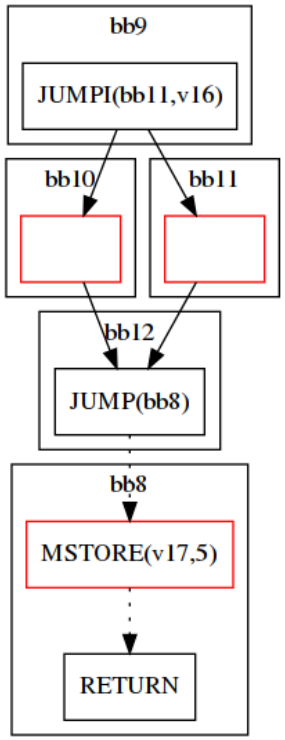


Figure: First Idea

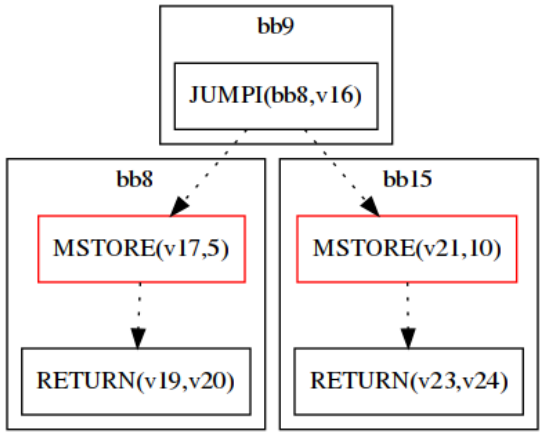


Figure: Second Idea

Stack Abstraction: Second Idea

1. Replicate the Basic Blocks from the CFG with empty content
2. Recursive instantiation
    3. Transform the Operations into Instructions
    4. Pass the stack to successors

Stack Abstraction: Second Idea

1. Replicate the Basic Blocks from the CFG with empty content
2. Recursive Roll–Out–Instantiation
   1. Differs the incoming stack to a possibly existing stack?
   2. If yes, then create a new BB with the same successors
   3. Transform the Operations into Instructions
   4. Pass the stack to successors
3. Remove Empty and Jump–only Basic Blocks

Evaluation: Usefulness of the second Idea

Bytes in Bytecode	BBs[2] with Operations	Operations	BBs with Instructions	Instructions
166	13	107	16	39
267	16	185	17	54
252	22	179	32	72
313	18	217	19	67

[2] Erays: Average Smart Contract contains 100 Basic Blocks with an average of 15 Instructions each

Stack Abstraction Algorithm

1. Replicate the Basic Blocks from the CFG as empty structure
2. Recursive instantiation

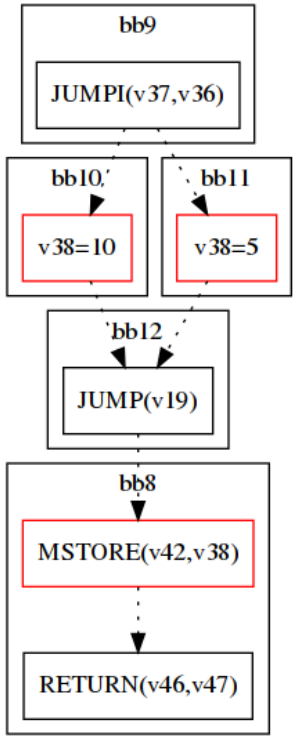
    2. Transform the Operations into Instructions
    3. Store the state of the stack

Stack Abstraction Algorithm

1. Replicate the Basic Blocks from the CFG as empty structure
2. Abstract for each bb
    1. Retrieve the stack from each predecessor and merge (same variable)
    2. Transform the Operations into Instructions
    3. Store the state of the stack

Stack Abstraction Algorithm

1. Replicate the Basic Blocks from the CFG as empty structure
2. Abstract for each bb
   1. Retrieve the stack from each predecessor and merge (same variable)
   2. Transform the Operations into Instructions
   3. Store the state of the stack



Evaluation Elements

- Operators (e.g. $+, *, \ll$)

Evaluation Elements

- Operators (e.g. $+, *, \ll$)
- Control Structures
  - if-else
  - loops
  - return, break, continue

Evaluation Elements

- Operators (e.g. $+$,$*$,$\ll$)
- Control Structures
  - if-else
  - loops
  - return, break, continue
- Datatypes (e.g. int, int8, arrays, struct, mapping)
  - fixed length
  - dynamic length

Evaluation Elements

- Operators (e.g. $+$,$*$,$\ll$)
- Control Structures
  - if-else
  - loops
  - return, break, continue
- Datatypes (e.g. int, int8, arrays, struct, mapping)
  - fixed length
  - dynamic length
- Functions (local, global)

Evaluation Items

ID	Name	CFG	Optimized	LOC	Bytes
1	1f_if	yes	yes	5	166
2	2f	yes	yes	6	267
3	2f_if	yes	yes	8	212
4	2f_ifif	yes	yes	15	252
5	2f_for_break	yes	no	11	227
6	2f_doWhileIf	yes	no	12	219
7	2f_ifWhile	yes	yes	11	243
8	1f_3-level-if	yes	yes	14	234
9	1f_string	yes	no	3	275
10	1f_if_string	yes	no	6	385
11	1f_string_concat	no	no	8	818
...	...	...	...	...	...
24	blind_auction	no	no	52	3066

Table: List of test examples. Displays whether the Base CFG and the optimized CFG can process the test correctly. Lines of code (LOC) and the number of Bytes in the Bytecode indicate the size of the test.

Other approaches

Decompiler (Solidity):

1. Porosity: Decompiler (Solidity)
2. Eveem: Decompiler (Solidity)

Security Analysis Frameworks:

1. Erays: Optimized CFG, Solidity
2. EthIR (Oyente): CFG, Rule–based Instructions
3. TheEther: CFG, Generates Exploits
4. Vandal: Opcode Representation, BB CFG